Exploiting Template Metaprogramming to Customize an Object-Oriented Operating System

S. Pinto, T. Castro, J. Mendes, S. Lopes, M. Ekpanyapong and A. Tavares
Centro Algoritmi - University of Minho
{sandro.pinto, tiago.castro, jose.mendes, sergio.lopes, adriano.tavares}@algoritmi.uminho.pt, mongkol@ait.ac.th

***Abstract*—Nowadays, the growing complexity of embedded systems demands for configurability, variability and reuse. Conditional compilation and object-orientation are two of the most applied approaches in the management of system variability. While the former increases the code management complexity, the latter leverages the needed modularity and adaptability to simplify the development of reusable and customizable software at the expense of performance and memory penalty. This paper shows how C++ TMP (Template Metaprogramming) can be applied to manage the variability of an object-oriented operating system and at the same time get ride out of the performance and memory footprint overhead. In doing so, it will be statically generated only the desired functionalities, thus ensuring that code is optimized and adjusted to application requirements and hardware resources.**

***Index Terms*—C++ Template Metaprogramming, Object-Oriented Operating System Design, Real-time Operating System, Variability and Configurability management, Feature Model.**

I. Introduction

Embedded systems are relevant for almost every market sector and represent a huge part of innovation in today's technology. Focus on embedded systems design has been driven mainly by (i) the high and fast penetration in products and services promoted by the integration of networks, operating systems and database capabilities, and (ii) the increasing computational performance for area and power-aware embedded processors. One indicative of these trends is that in recent years, approximately 98% of microprocessors annual production was aimed at embedded systems [1]. These trends led to several specific issues and challenges in design new embedded systems, where bare-metal embedded software development is no longer affordable due to a tremendous pressure on time-to-market, increased time and effort development, and poor end-product quality. So, the focus has been directed to new embedded systems design methodologies that are able to keep the pace of software productivity with the speed of hardware innovation.

Due to the growing complexity and demanding flexibility of current systems, the integration of operating systems into the embedded systems software stack makes the system development easier, faster and safer. However, it also claims for an increasing demand for configurability, variability and reuse strategies [2], to better accommodate new embedded systems design methodologies. Typically, monolithic operating system models do not fit the requirements and limitations of embedded systems since they attempt to maximize the number of supported platforms and offered features, resulting in increasing resources consumption. Therefore, the focus turns to microkernel-based RTOS (Real-Time Operating Systems) tailored to the processor architecture and application requirements and constraints. Most of the applied approaches in small embedded systems domain manage RTOS variability using conditional compilation or object-orientation. The former paradigm makes the code management harder [3] while the latter one provides the modularity and adaptability needed to simplify the task of developing reusable and customizable software. However, object-orientation in embedded systems domain incurs a huge abstraction penalty, if based on weighty languages dialects like dynamic polymorphism and multiple inheritance, features that increase overhead in space and performance [4], [5].

This paper proposes the use of C++ TMP [6], [7] as a methodology for managing the variability of an object-oriented RTOS and generate only the desired functionalities, ensuring that the code is optimized and adjusted to application requirements and hardware resources. Section II briefly presents some similar works. Section III introduces ADEOS, as well as its feature diagram, in order to identify the common and variable features on the operating system. Section IV describes ADEOS domain design and implementation, upgrading and refactoring. Section V describes the used evaluation process based on performance and memory footprint. Section VI concludes.

II. Related Work

OS adaptation in embedded systems is not new and platforms to create Application Specific Operating Systems (ASOS) include examples such as: Koala, eCOS, PURE and CiAO. Their adaptation mechanisms encompass (i) component-based development in Koala [8], (ii) C preprocessor directives in eCOS and Linux [9], (iii) object orientation in PURE and (iv) Aspect Oriented Programming (AOP) [10], [11] in CiAO, used to configure operating systems oriented to resource-constrained embedded devices. These approaches can, in a broad sense, be categorized as software product lines (SPL) and they have already been applied in other domains, using different technologies. In [12] efforts have been done to identify the Linux variability and make it usable as ASOS.

Variability can be managed using several technologies. The simplest way to achieve it in C/C++ is through conditional compilation (Linux has its configurability system based on it, using the so called kconfig model [13]). This method does not

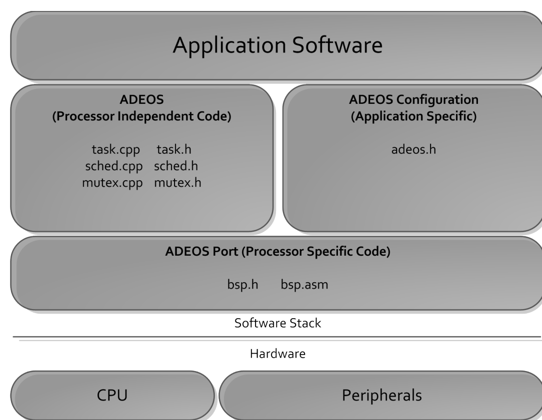

Fig. 1: ADEOS software architecture

induce overhead and has fine-grained granularity, but generates code difficult to understand, maintain, evolve and reuse [14]. Patches [15] are also heavily used on Linux, having the additional advantage that base code is not polluted. However, variants are maintained in an awkward format, worsening the problems of the previous technique. Taking into account patch disadvantages, semantic patch was proposed, trying to perform only localized changes that are later applied into the code. Coccinelle is one such example, using a notation that resembles the patch language but it did not perform very well due to its semantic biasing towards code evolution [16].

III. ADEOS Domain Analysis

To tackle the increasing complexity of current embedded systems and offering the desired congurability, variability and reusability, software technologies like generative programming [6], [17] and model driven development [18] will be combined with SPL engineering to bridge the abstraction gap between domain modeling and feature modeling. Before starting with the domain analysis to identify the commonality and variability as well as the relationship among RTOS components and features in the variability model, ADEOS will be shortly introduced.

A. A Decent Embedded Operating System

ADEOS (A Decent Embedded Operating System) is an object-oriented operating system developed by Michael Barr [19]. It was written in C++, and designed to be deployable to embedded systems with scarce resources. The task manager is responsible to create, add, remove and run tasks in a multitasking environment. The scheduler policy implements a fixed-priority and pre-emptive scheduling. The kernel code is small (less than 1000 lines) and most of them are architecture independent; only the context switch code is targeted to the 80188 processor. Among the few existing object-oriented operating systems, ADEOS was chosen because the main target is small embedded systems. ADEOS was refactored and extended with new features and also to follow a microkernel-based RTOS model. A porting to MCS-51 (8051 Microcontroller Family) family was carried out due its severe resource constraints microcontrollers. Fig. 1 presents ADEOS software architecture.

From Fig. 1, one can realize where ADEOS porting effort to MCS-51 should be focused on. Processor-dependent code are distributed in header and code file bsp.h and bsp.asm, respectively. The header file specifies the structure of a task context (Listing 1) and the low-level function prototypes (Listing 2). The former represents the microprocessor state to a specific task, which includes the address of the current program instruction, the microcontroller registers and flags, and the address of internal and external stack. The function prototypes declare the routines responsible to initialize the context of a specific task, and switch among them.

```
typedef struct context
{
        unsigned char PC_H, PC_L;
        unsigned char A, B;
        unsigned char IE;
        unsigned char DPL, DPH;
        unsigned char R0, R1, R2, R3, R4, R5, R6, R7;
        unsigned char PSW;
        unsigned char SP;
        unsigned char XSP_H, XSP_L;
} Context;
```

Listing 1: Machine state

```
extern "C"
{
        extern void idle(void);
        extern void contextInit(Context * pContext,
                                void (*run)(Task * pTask),
                                Task * pTask);
        extern void contextSwitch(Context * pOldContext,
                                Context * pNewContext);
}
```

Listing 2: Processor-dependent low-level functions signatures

The implementation or definition of processor-dependent low-level functions are given in the bsp.asm file. Listing 3 presents a snippet of *contextInit* function, responsible to initialize the *PC_H* and *PC_L* attributes from task context structure, which represents the address of task start-up routine.

```
;Get the pointer to context
MOV DPH, 3; Load pContext_H into DPH
MOV DPL, 2; Load pContext_L into DPL
;Initialize the pointer to startup routine
MOV A, 5; A = pFunc_H
MOVX @DPTR, A; pContext−>PC_H = pFunc_H
INC DPTR; point to pContext−>PC_L
MOV A, 4; A = pFunc_L
MOVX @DPTR, A; pContext−>PC_L = pFunc_L
```

Listing 3: Initialization of task start-up routine address

B. Feature Model

Feature diagram is a visual representation of feature model [20] and it represents a set of features organized hierarchically, starting at the root node that represents the system concept, and all possible links between all its nodes. The model allows the management of system commonality and variability,

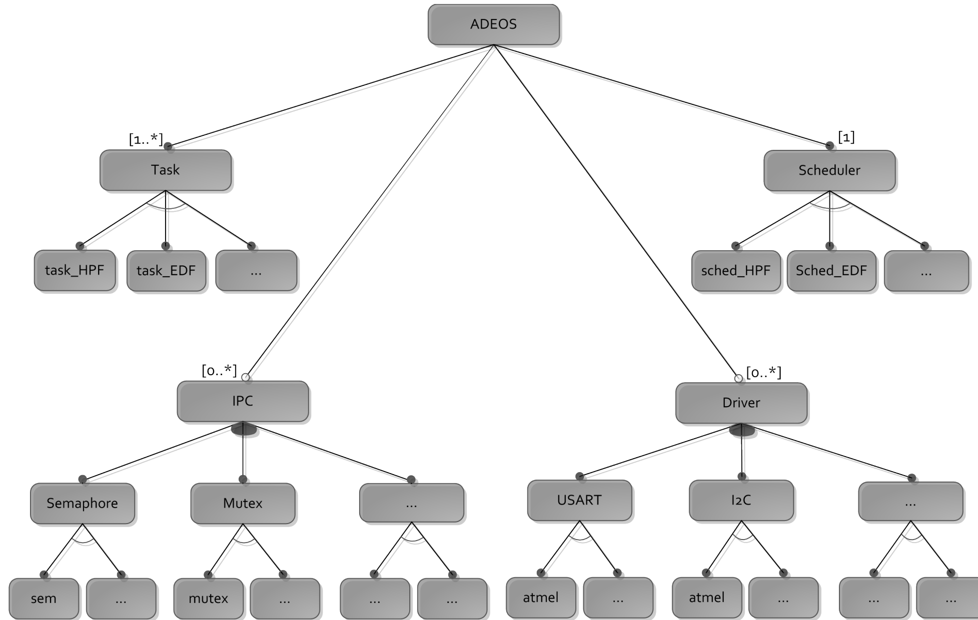

Fig. 2: ADEOS feature diagram

disregarding the implementation mechanism. Four kinds of features can be represented:

- *Mandatory features*: The system must have, mandatorily, some features. Mandatory features are pointed to by simple edges ending with a filled circle.
- *Optional features*: The system may, or not, have some features. Optional features are pointed to by simple edges ending with an empty circle.
- *Alternative features*: The system can have only one feature at a time. Alternative features are pointed to by edges connected by an arc.
- *Or-features*: The system may have a combination of features. Or-features are pointed to by edges connected by a filled arc.

Fig. 2 depicts a feature diagram showing variability in ADEOS and explicit features that can be managed in order to be able to set scheduling policy, number and type of IPC (Inter-Process Communication), and even different device drivers, according to the application requirements.

The root node represents the concept (ADEOS) which is composed by four features: *Task*, *IPC*, *Driver* and *Scheduler*. *Task* feature has cardinality [1..*], which means that ADEOS runs, at least one task (i.e., the idle task). However, the task feature is alternative, because TCB (Task Control Block) is characterized by the scheduler. For example, the task is characterized by priority if uses a high-priority-first (HPF) scheduler, or by deadline if uses an earliest deadline first (EDF) scheduler. *Scheduler* has cardinality [1], namely, ADEOS has by default only one scheduler and like task, is also alternative, which means that only one policy can be included in a specific configuration. On the other hand, *IPC* and *Driver* features has cardinality [0..*], which represents that they are optional features. For example, the *Driver* feature is only necessary to include if it is required to address some peripheral. Likewise, the *IPC* feature is only included if tasks need synchronization and communication. In this sense, these two features are or-features, but they also represent alternative variability. For example, it can be necessary to address an SPI (Serial Peripheral Interface Bus) and USART (Universal Synchronous/Asynchronous Receiver/Transmitter) device at the same time. However, the peripheral specification can be only one. For instance, in MCS-51 different manufacturers (Atmel, Texas, and so on) have specific peripherals implementations. So, different versions of a specific driver can exist to target different peripherals implementation. But only one can be used in a given configuration.

IV. ADEOS Domain Design and Implementation

Taking into account that most of small embedded applications variability can be managed at compile time, generative programming can be applied for automatic selection and assembly of components based on a configuration knowledge that will be implemented using C++ TMP as the generator. Therefore, allowing system configurations and simultaneously preventing undesired and unnecessary overhead induced by the previously mentioned approaches.

In spite of the modular architecture leveraged by encapsulating each operating system component into a class, ADEOS follows a monolithic model that later will be changed to a microkernel model by extending and adapting the original IPC mechanisms. This paper will focus on refactoring original ADEOS classes into customizable and configurable artifact templates encoded in C++ TMP to support the original monolithic model. Only the artifact templates related to scheduler and task classes will be discussed as the others will be similarly refactored. Listing 4 presents the scheduler template

declaration with the template parameter specified later as the selected scheduler given by a desired configuration.

```
//Scheduler template declaration
template <typename schedType> class schedManager;
```

Listing 4: Scheduler class template declaration

Listings 5 and 6 show the artifact templates representing the scheduler component. During the template instantiation process, the compiler selects the template that best matches the given template argument (i.e., class name that implements the selected scheduler policy). It was necessary to specialize as many templates as the number of schedulers, since different scheduler implementations can have different methods designations, arguments and behaviours.

```
//Specialization for HPF scheduler
template <>
class schedManager<sched_HPF> {
        private:
                sched_HPF sched_;
        public:
                void start() {
                        sched_.start_HPF();
                }
                void schedule() {
                        sched_.schedule_HPF();
                }
                void addTask(Task ∗ pTask) {
                        sched_.addTask_HPF((pTask);
                }
                void enterIsr() {
                        sched_.enterIsr_HPF(();
                }
                void exitIsr() {
                        sched_.exitIsr_HPF(();
                }
                ...
};
```

Listing 5: HPF scheduler specific template definition

```
//Specialization for EDF scheduler
template <>
class schedManager<sched_EDF> {
        private:
                sched_EDF sched_;
        public:
                void start() {
                        sched_.start_EDF();
                }
                void schedule() {
                        sched_.schedule_EDF();
                }
                void addTask(Task ∗ pTask) {
                        sched_.addTask_EDF(pTask);
                }
                void enterIsr() {
                        sched_.enterIsr_EDF();
                }
                void exitIsr() {
                        sched_.exitIsr_EDF();
                }
                ...
};
```

Listing 6: EDF scheduler specific template definition

Each scheduler is implemented as a module in an individual header file, and it is necessary to specialize as many dedicate templates as needed scheduler polices. Since the class template methods are implemented in class definition, they are implicitly declared as *inline*, speeding-up performance as no calls are done during execution.

For clarity, in Listing 7 *typedef* is used to add a template designation and also to specify the chosen template that should be instantiated. Finally, Listing 8 shows the customization transparency offered by such parametric polymorphism, as no change needs to be done on ADEOS original main function. Simply replacing the template parameter *sched_HPF* to *sched_EDF* on Listing 7, it is possible to run the same code in Listing 8 with a different scheduler policy.

```
typedef scheduleManager<sched_HPF> sched;
```

Listing 7: Scheduler template specification

```
sched os; //Declare scheduler template variable
...
int main() {
        os.addTask(&Idle); //Add idle Task to readyList
        os.addTask(&taskA); //Add Task to readyList
        os.addTask(&taskB); //Add Task to readyList
        os.start(); //Start OS
        return 0;
}
```

Listing 8: Example using scheduler template

The task manager class is refactored following the same approach with only slight differences. Note that the task component depends on the chosen scheduler policy and will present variability at task specificity as well as at list level (several lists will be used, one to represent the ready tasks and several others representing blocked tasks). Listing 9 presents an enumeration specifying the task lists type, and the task manager template declaration. Thus, the template arguments should specify the list and task type.

```
//List Type
enum TypeList { readyList, waitList };
//Task manager template declaration
template <TypeList list, typename taskType> class taskManager;
```

Listing 9: Task manager class template declaration

Listings 10 and 11 present the artifact templates for task manager class. The first parameter specifies the list type, while the second one is the class name that implements the task type. Similarly to scheduler, each task class is implemented as a module in an individual header file. This way, it is necessary to specialize as many dedicated templates as the list and task types.

```
template <>
class taskManager <readyList, TaskList_HPF> {
    private:
        TaskList_HPF readyList_;
    public:
        void insert(task_HPF ∗ pTask) {
            readyList_.insertHPF(pTask);
        }
        task_HPF ∗ remove(task_HPF ∗ pTask) {
            return readyList_.removeHPF(pTask);
        }
        ...
```

```
};

template <>
class taskManager <waitList, TaskList_HPF> {
    private:
        TaskList_HPF waitList_;
    public:
        void insert(task_HPF * pTask) {
            waitList_.insertHPF(pTask);
        }
        task_HPF * remove(task_HPF * pTask) {
            return waitList_.removeHPF(pTask);
        }
        ...
};
```

Listing 10: HPF task manager specific template definition

```
template <>
class taskManager <readyList, TaskList_EDF> {
    private:
        TaskList_EDF readyList_;
    public:
        void insert(task_EDF * pTask) {
            readyList_.insert2(pTask);
        }
        task_EDF * remove(task_EDF * pTask) {
            return readyList_.remove2(pTask);
        }
        ...
};

template <>
class taskManager <waitList, TaskList_EDF> {
    private:
        TaskList_EDF waitList_;
    public:
        void insert(task_EDF * pTask) {
            waitList_.insert2(pTask);
        }
        task_EDF * remove(task_EDF * pTask) {
            return waitList_.remove2(pTask);
        }
        ...
};
```

Listing 11: EDF task manager specific template definition

To initialize the task list of a given application is defined a generic template (Listing 12) and a specialized one (Listing 13) to implement the rolling cycle, and the stop condition, respectively. The generic class template only accepts one template parameter, that is a static list of tasks (Typelist [21]). The variable *tail* is a recursive template that uses the tail of the current list as a template parameter. The recursion is stopped with the explicit specialization for the *NullType*.

```
//Task template declaration
template<typename TList> class CTask;

//Typelist template definition
template<typename T, typename U>
class CTask< Typelist<T,U> > {
        private:
                T first_;
                CTask<U> tail_;
                Task_p * par;
        public:
                void init(Task_p * par_) {
                    first_.init(par_);
                    tail_.init(par_->next);
                }
};
```

Listing 12: Typelist task template definition

```
//NullType template definition
template<>
class CTask<NullType> {
        public:
                void init(Task_p * par_) { }
};
```

Listing 13: NullType class template definition

Finally, Listing 14 shows a declaration (*typedef*) that allows the creation of a Typelist with 2 types (tasks), the template instantiation and tasks initialization. The *init* method is called once, since the recursion is intrinsic to the template. In this Listing, *Task* is the designation of the class tasks and *Results* is the designation of the tasks' static list (Typelist).

```
//Typelist instantiation
typedef MakeTypelist<Task,Task>::Result list;
CTask<list> tasks;
...
//Initialize Tasks
tasks.init(parameters);
```

Listing 14: Tasks initialization example using Typelists

V. EVALUATION

To simplify the system evaluation only a limited number of possible configurations were tested to reflect variability at only two levels, instead of following each new feature added to a given configuration. Thus, only 32 different congurations were tested and measurement data gathered to compare the variability management using C++ TMP and dynamic polymorphism. For the former approach a predefined configuration that implements an RTOS with a preemptive priority-based scheduler, with mutex and two device drivers (USART and SPI) targeting Atmel 8051 family (AT89C51) was chosen. The system runs two tasks: (i) sending a character using serial port; and (ii) communicates with an SPI slave device. The character is sent every two seconds, and the SPI communication is established every five seconds. Data related to performance (i.e., execution time), memory footprint and code management (lines of code and number of classes) were collected.

A. Performance and Memory Footprint

The performance and memory footprint results reflects, respectively, the execution time in clock cycles and code memory in bytes required to run system tasks with different implementations based on C++ DP (Dynamic Polymorphism) and C++ TMP. To get the execution time the *IAR Embedded Workbench 8051 debugger* was used. This evaluation does not take into account the effective time spent to send the character through the serial port and to communicate with SPI slave device. To measure the code size the *Atmel Flip* software was used. Table I presents the achieved results.

The results show that C++ TMP implementation has lower execution time and memory footprint than C++ DP. The

TABLE I: Performance and Memory Footprint Results

Implementation	Execution Time (clock cycles)	Code Memory (bytes)
C++ DP	34728	12987
C++ TMP	27851	7958
Comparison	**-20%**	**-40%**

TABLE II: Code Management Results

Implementation	Lines of Code (LOC)	Number of Classes (NOC)
C++ DP	1232	29
C++ TMP	1325	41
Comparison	**+8%**	**+41%**

achieved improvements are 20% and 40% in performance and memory footprint, respectively. The pointed reason is that TMP code is optimized to a specic conguration and is free from unused instructions and indirect calls. Furthermore, some more experiments were carried out with tree levels in variability and the results show that C++ TMP implementation can reduce execution time and memory footprint around 25% and 55%, respectively.

B. Code Management

The code management results show the number of lines of code (LOC), excluding comments and blank lines, and the number of classes (NOC) required to implement all RTOS features. To get code management metrics the *Understand* software by Scientic Toolworks was used. Table II presents the achieved results. From Table II one can conclude that LOC is identically in both implementations. However, relatively to NOC, C++ TMP implementation has a higher value. This means that TMP code offers higher modularity and encapsulation than code implemented using dynamic polymorphism, which is better to manage, maintain and reuse.

VI. Conclusion and Future Work

This paper described how template metaprogramming can be exploited to statically configure and customize, an object-oriented operating system. Since this approach uses an object-oriented programming language, it simplifies code management task. Furthermore, since this advanced programming technique does not use none of those weighty C++ dialects and only compile a specific configuration, the code is optimized and overhead free. The results showed that C++ TMP approach presents some advantages related to performance and memory footprint when compared to traditional object-oriented approach.

Under development are several tasks such as: (i) upgrading ADEOS with more features, to increase system variability and following a microkernel model; (ii) promoting the passage from development for reuse to the development with reuse by combining MDD (Model-Driven Development) approach with feature models and domain artifact template models. Mainstream tools like Eclipse and Xpand are used. At the end, a plain C implementation of ADEOS will be proposed, using conditional compilation to manage variability. This way, it will be possible to evaluate TMP implementation against a C implementation, and discuss exhaustively the achieved results in terms of performance and memory footprint.

VII. Acknowledgements

This work is supported by FEDER through COMPETE and national funds through FCT Foundation for Science and Technology in the framework of the project FCOMP-01-0124-FEDER-022674.

References

[1] D. Tennenhouse, “Proactive computing,” *Communications of the ACM*, vol. 43, no. 5, pp. 43–50, 2000.

[2] AUTOSAR, “Requirements on Operating System,” GbR, Automotive Open System Architecture, Tech. Rep., 2006.

[3] Y. Hu, E. Merlo, M. Dagenais, and B. Lague, “C/C++ conditional compilation analysis using symbolic execution,” *Proceedings of the International Conference on Software Maintenance (ICSM’00)*, 2000.

[4] P. Plauger, “Embedded C++: An Overview,” *Embedded Systems Programming*, 1997.

[5] D. Herity, “C++ in embedded systems: Myth and reality,” *EE Times India*, 1998.

[6] U. E. K. Czarnecki, *Generative Programming: Methods, Tools, and Applications*, 1st ed., Addison-Wesley Professional, Ed., 2000.

[7] A. G. D. Abrahams, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, 1st ed., A.-W. Professional, Ed., 2005.

[8] E. I. Kaur, P. K. Suri, and E. A. Verma, “Characterization and Architecture of Component Based Models,” *International Journal of Advanced Computer Science and Applications*, vol. 1, no. 6, pp. 66–71, 2010.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “Variability modeling in the real: a perspective from the operating systems domain,” *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 73–82, 2010.

[10] D. Lohmann, “Pure Embedded Operating Systems-CIAO*,” *International Workshop on Operating Systems Plataforms for Embedded Real-Time Applications*, 2006.

[11] D. Lohmann, W. Hofer, W. Schröder-Preikschat, and O. Spinczyk, “Aspect-aware operating system development,” *Proceedings of the tenth international conference on Aspect-oriented software development - AOSD ’11*, p. 69, 2011.

[12] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “The variability model of the linux kernel,” *Fourth International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010)*, 2010.

[13] J. Sincero and W. Schröder-Preikschat, “The linux kernel configurator as a feature modeling tool,” *Proceedings of the Workshop on Analyses of Software Product Lines (ASPL)*, 2008.

[14] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, “Can we refactor conditional compilation into aspects?” *Proceedings of the 8th ACM international conference on Aspect-oriented software development - AOSD ’09*, p. 243, 2009.

[15] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files With Gnu Diff and Patch*, 1st ed. Network Theory Ltd, 2003.

[16] F. Armand, J. Berniolles, J. L. Lawall, and G. Muller, “Automating the Porting of Linux to the VirtualLogix Hypervisor using Semantic Patches,” *Embedded Real Time Software 2008 (ERTS 2008)*, 2008.

[17] K. Czarnecki, “Generative programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models,” Ph.D. dissertation, Ilmenau, 1998.

[18] A. Kleppe, J. Waremer, and W. Bast, *MDA Explained The Model Driven Architecture: Practice and Promise*, 1st ed. Addison-Wesley Professional, 2003.

[19] M. Barr, *Programming Embedded System in C and C++*, 1st ed., P. O. Reilly, Ed., 1999.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” Carnegie-Mellon University Software Engineering Institute, Tech. Rep. November, 1990.

[21] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, 1st ed., Addison-Wesley Professional, Ed., 2001.